Homework 6 Introduction to Big Data Systems

Christoffer Brevik

October 29, 2024

1 Word Count

1.1 How to Run the code

Running my WordCount program can be done by going into the `./2024403421/hw6_src` folder in the computational node, or the root folder of my delivery file. From here you can run the command:

```
python3 word_count.py /hw6_data/pg100.txt
```

Below is the code that this command will run, an explanation of this and the results you should expect by running this code

1.2 My code

To solve this task I expanded the `word_count.py` to be the following:

```
import re
import sys
from pyspark import SparkConf, SparkContext
import time

if __name__ == '__main__':
    conf = SparkConf()
    sc = SparkContext(conf=conf)
    sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal
    lines = sc.textFile(sys.argv[1])
    top_i_words = 10

    first = time.time()

    # We split the lines into words
    words = lines.flatMap(lambda line: re.split(r'[^\w]+', line))

    # We now count every word
    word_counts = words.countByValue()
    top_words = sorted(word_counts.items(), key=lambda x: -x[1])[:top_i_words]

    # Print the top i words
    print(f"Top {top_i_words} words:")
    for word, count in top_words:
        print(f"({repr(word)}, {count})")

    last = time.time()

    print("Total program time: %.2f seconds" % (last - first))
    sc.stop()
```

1.3 Explanation of code

After importing the relevant libraries, we set up *PySpark*. Here we specify that we only want Spark to log errors and not warnings. Afterwards we import the specified txt-file as a RDD (Resilient Distributed Dataset) and specify how many of the top words the program should display:

```
conf = SparkConf()
sc = SparkContext(conf=conf)
sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal

lines = sc.textFile(sys.argv[1])
top_i_words = 10
```

We begin by recording the program's start time to measure execution duration. Next, we split each line into individual words using the `flatMap` function, which applies a lambda function with `re.split` to separate words based on the regex pattern provided in the assignment's ReadMe:

```
first = time.time()

# We split the lines into words
words = lines.flatMap(lambda line: re.split(r'[^\w]+', line))
```

It is important to note here that you can replace "line" with "line.lower()" to get the count of all words ignoring capitalization. But doing this made the answers not match what was considered correct in the task, so my final application does not implement this

Afterwards, we count each word's occurrences using the `countByValue()` function on the `words` dataset. This method returns a dictionary where each key is a unique word and each value is the word's count. Following this, we use `sorted()` to order the dictionary items by frequency, in descending order. From here we can just slice the `top_i_words`, getting the i most used words:

```
# We now count every word
word_counts = words.countByValue()
top_words = sorted(word_counts.items(), key=lambda x: -x[1])[:top_i_words]
```

Afterwards we can iterate over the sliced words and display their counts in the format specified in the Readme, printing this to the console:

```
# Print the top i words
print(f"Top {top_i_words} words:")
for word, count in top_words:
    print(f"({repr(word)}, {count})")
```

Finally, we calculate and print the total runtime by subtracting the starting time, `first`, from the end time, `last` and stop the PySpark instance:

```
last = time.time()
print("Total program time: %.2f seconds" % (last - first))
sc.stop()
```

1.4 Results

By running the program we get the following output:

```
Top 10 words:
('', 172754)
('the', 25743)
('I', 23849)
('and', 20184)
('to', 17424)
('of', 17301)
('a', 13974)
('you', 12901)
('my', 11411)
('in', 11384)
Total program time: 4.13 seconds
```

This matches the correct answer stated in the ReadMe.

2 PageRank

2.1 How to Run the code

Running my PageRank-program can be done by going into the `./2024403421/hw6_src` folder in the computer node or the root of my delivery file. Here you have the option on running my code with the dynamic setting turned on or off, explained in 1.3 (both returns the same results, but have different runtime). While the assignment only mentioned that my grade would be based on the larger dataset, I've also added the command I used to get my results for the smaller dataset in 2.1.3 so that my results can be replicated.

2.1.1 Dynamic Setting turned on (Recommended)

```
python3 page_rank.py /hw6_data/full.txt True
```

2.1.2 Dynamic Setting turned off

```
python3 page_rank.py /hw6_data/full.txt False
```

2.1.3 Code used for my Results for small.txt

```
python3 page_rank.py /hw6_data/small.txt False
```

Below is the code that this command will run, an explanation of this and the results you should expect by running this code

2.2 My Code

To solve the PageRank task, I expanded the given `page_rank.py` to be the following:

```
import sys
from pyspark import SparkConf, SparkContext
import time

if __name__ == '__main__':
    conf = SparkConf()
    sc = SparkContext(conf=conf)
    sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal
    file_path = sys.argv[1]
    lines = sc.textFile(file_path)

    # Parameters
    damping = 0.8
    num_iterations = 50
    top_i_nodes = 5
    dynamic = sys.argv[2]

    first = time.time()

    # Parse the lines into (source, destination) pairs and remove duplicates
    edges = lines.map(lambda line: tuple(map(int, line.split()))).distinct()

    # We estimate the amount of nodes
    if(dynamic):
        # This method takes aprox. 0.5s longer but is dynamic
        nodes = edges.flatMap(lambda edge: edge).distinct()
        n = nodes.count()
    else:
        # This method is faster but not dynamic
```

```
    n = 100 if "small" in file_path else 1000 if "full" in file_path else None

# Create an adjacency list as (node, [neighbors])
adj_list = edges.groupByKey().mapValues(list).cache()

# Initialize each node's PageRank value
page_ranks = adj_list.mapValues(lambda _: 1.0 / n)

for i in range(num_iterations):
    # Broadcast the adjacency list for efficient access
    adjacency_broadcast = sc.broadcast(adj_list.collectAsMap())

    # Compute contributions for each node's neighbors
    contributions = page_ranks.flatMap(lambda node_rank: [
        (neighbor, node_rank[1] / len(adjacency_broadcast.value.get(node_rank[0], [])))
        for neighbor in adjacency_broadcast.value.get(node_rank[0], [])
    ])

    # Aggregate contributions and calculate new PageRank values
    page_ranks = contributions.reduceByKey(lambda a, b: a + b).mapValues(
        lambda rank: (1 - damping) / n + damping * rank
    )

# Get the top 5 nodes with the highest PageRank scores
highest = page_ranks.takeOrdered(top_i_nodes, key=lambda x: -x[1])

# Print the top 5 nodes
print("Top 5 nodes with highest PageRank scores:")
for node, score in highest:
    print(f"Node {node}: {score}")

last = time.time()
print("Total program time: %.2f seconds" % (last - first))
sc.stop()
```

2.3 Explanation of Code

As with WordCount, the program begins by setting up *PySpark* and configuring it to not log warnings. The specified input file is loaded as an RDD, and the parameters for the PageRank algorithm are initialized. As stated in the ReadMe, we have a d-value of 0.8 and use 50 iterations. We also specify that we only want the top 5 nodes. The last variable, `dynamic`, is used to estimate the amount of nodes, which is explained later:

```
conf = SparkConf()
sc = SparkContext(conf=conf)
sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal
lines = sc.textFile(file_path)

# Parameters
damping = 0.8
num_iterations = 50
top_i_nodes = 5
dynamic = False
```

The program then begins by recording the initial timestamp in `first`. Next, the lines of the file are parsed into source-destination pairs representing the edges of the graph. We also remove the duplicates using the *.distinct()*-function, which reduces the amount nodes we have to go trough:

```
first = time.time()

edges = lines.map(lambda line: tuple(map(int, line.split()))).distinct()
```

To estimate PageRank we need the total amount of nodes, `n`. Here I've created two different versions, choesn by the user using the `dynamic`-variable. If `dynamic = True` then we will count the amount of nodes in our dataset, making the algorithm dynamic. This operation is *O(n)* and takes aprox. 0,5s more than the static version on the "full.txt". If `dynamic = False`, then we only check which of the two files we are reading and sets `n` to the corresponding values stated in the ReadMe. This was only made this way as I wanted to experiment with making the algorithm even faster:

```
# We estimate the amount of nodes
if(dynamic):
    # This method takes aprox. 0,5s but is dynamic
    nodes = edges.flatMap(lambda edge: edge).distinct()
    n = nodes.count()
else:
    # This method is faster but not dynamic
    n = 100 if "small" in file_path else 1000 if "full" in file_path else None
```

Afterwards we create an adjacency list where each node is mapped to a list of its neighbors. PageRank values for each node are initialized to $\frac{1}{n}$, where `n` is our number of nodes:

```
# Create an adjacency list as (node, [neighbors])
adj_list = edges.groupByKey().mapValues(list).cache()

# Initialize each node's PageRank value
page_ranks = adj_list.mapValues(lambda _: 1.0 / n)
```

After setting the initial PageRank values, the program performs 50 iterations to refine the PageRank scores of each node. The program calculates the contribution each node makes to its neighbors by dividing its PageRank score by the number of neighbors, which is then distributed to the neighboring nodes. These contributions are aggregated for each node, and the PageRank values are updated according to the PageRank formula, incorporating the damping factor. In each iteration, the adjacency list is broadcasted to each worker to optimize data access and avoid redundant computations:

```
for i in range(num_iterations):
    adjacency_broadcast = sc.broadcast(adj_list.collectAsMap())
    contributions = page_ranks.flatMap(lambda node_rank: [
        (neighbor, node_rank[1] / len(adjacency_broadcast.value.get(node_rank[0], [])))
        for neighbor in adjacency_broadcast.value.get(node_rank[0], [])
    ])
    page_ranks = contributions.reduceByKey(lambda a, b: a + b).mapValues(
        lambda rank: (1 - damping) / n + damping * rank
    )
```

Finally, the top nodes by PageRank score are printed, along with their scores. As no "correct" format was given for the nodes, I just displayed them in a column:

```
highest = page_ranks.takeOrdered(top_i_nodes, key=lambda x: -x[1])
print("Top 5 nodes with highest PageRank scores:")
for node, score in highest:
    print(f"Node {node}: {score}")
```

Finally, we calculate and print the total runtime by subtracting the starting time, `first`, from the end time, `last` and stop the PySpark instance:

```
last = time.time()
print("Total program time: %.2f seconds" % (last - first))
sc.stop()
```

2.4 Results

To check that my program works the same way as described in the readme, I checked it on the *small.txt*. Here I've only reported with the dynamic-setting turned off. Here we see that 71 had the correct value of aprox. 0.0190909:

2.4.1 small.txt, Dynamic = True

```
Top 5 nodes with highest PageRank scores:
Node 71: 0.019090928823281568
Node 32: 0.01650228070983296
Node 31: 0.016040052755192998
Node 33: 0.01526161330340896
Node 81: 0.015219911388697908
Total program time: 13.29 seconds
```

Furthermore, by running the program on the *full.txt* dataset we get the following results

2.4.2 full.txt, Dynamic = True

```
Top 5 nodes with highest PageRank scores:
Node 368: 0.0020288868721239453
Node 526: 0.0019387314593812125
Node 696: 0.0019226745721505072
Node 701: 0.0018952349791332532
Node 897: 0.0018450838839604762
Total program time: 14.10 seconds
```

2.4.3 full.txt, Dynamic = False

```
Top 5 nodes with highest PageRank scores:
Node 368: 0.0020288868721239453
Node 526: 0.0019387314593812125
Node 696: 0.0019226745721505072
Node 701: 0.0018952349791332532
Node 897: 0.0018450838839604762
Total program time: 13.68 seconds
```